Homework 4: Q2

Name: Waiwai Kim, waiwaiki

1 Part (a): Formalizing the problem

Here we represents ADFs as a graph $G = (V, E)$ where fans are represented as the vertices V of the graph G. The edges E of graph G represent friendship. i.e) edge E exists between (i, k) if i and k are friends. Seating is a one-to-one function that assigns V to possible p^2 seating.if $s(u) = (r, c)$ means that if the seat of u is at row r and column c. This is already presented in the week 6 recitation note.

Now let's formalize an admissible seating. There are three properties that an admissible seating needs to satisfies.

- Friends are able to talk to each other by being in the same row or the adjacent row. Column doesn't matter. If an edge (freindship) exists between i and k, and $s(i) = (r_i, c_i)$ (i sits on row r_i and column c_i) and $s(k) = (r_k, c_k)$, it must be the case that $|r_i - r_k| \leq 1$.
- At least one ADF must sit in the first row. $\exists u \in V$ such that $s(u) = (1, c)$.
- Distance compatibility. If an ADF a is x degree friend of b who sits on the front row, a must sit on $x + 1$th row. For example, if a is b's friend (1st degree friend), a sits on the second row. Similarly, if a is a b's friend of a friend (second degree friend), a sits on the third row. In graph $G(u, v)$, the distance compatibility between a and b is defined by distance $d(a, b)$ where distance is the shortest path from node a to node b. This can be figured out by BFS as we have seen Q1.

2 Part (b): Algorithm Idea

We will build an algorithm that produces an admissible seating based on BFS algorithm as we have seen in Q1. First we will convert the list of pairs into an adjacency list so that we can leverage the skeleton of the algorithm we have written for Q1.

First, let's start with how we want to structure distance compatibility which is the most restricting qualification to be an admissible seating. As it has been mentioned in the bullet point above, we can think of distance compatibility as the distance between the starting node and a connected node. Note that it only matters which row that a connected node goes to. On the flip side, which column that the connect node sits is irrelevant. We can sit a connected node anywhere in the correct row. Also we do not have to worry about overcrowding a row because we have p seatings in a row and we have total p ADFs. Running the BFS algorithm on the starting node will visit all the nodes that need to sit on the second row. Similar to Q1, we can use queue in order to implement this algorithm. Running the BFS algorithm on the nodes in the second row will visit all the nodes for the third row.

Secondly, ADF that sits on the first row represents the starting node in a graph. We have freedom on how to choose the first node since there are more than one admissible seating, and our algorithm needs to produce one valid admissible seating. For our algorithm, we will choose an ADF who has the most number of friends or the largest number of connected vertices. This rule implies that the starting node our algorithm chooses will have the largest number of ADFs sitting on the second row for any given candidate who can sit on the first row.

Thirdly, being a friend simply means that two ADFs have an edge between them. This is somewhat not restricting because we can sit a friend in the same row or an adjacent row.

Applying the BFS algorithm on an adjacency list that takes the three aforementioned points into an account will completes the entire seating if all nodes are connected (not necessarily connected to the starting node). However, our algorithm must sit an ADF or a group of ADFs who have absolutely no connection to previous groups that have just taken seats. We can do so by checking whether everyone has taken a seat and repeat the algorithm on unseated nodes.

3 Part (b): Algorithm Details

Note that everything presented in Algorithm Details is an actual Python code.

First, let's start with input to the algorithm. In Figure 1, we can see a few example of input: 1) a list of pairs or tuples in Python an d2) the number of ADFs represented as n. With this information, we can construct an appropriate theater. For example, if $n = 5$, the theater will be 5×5 theater. In implementation, we will be an array size of n^2.

Figure 1: ADF Seating - Input

```
def __init__(self):

    '''
    input_list = [(1,3), (1,2)]                 # input_list = list of tuple
    n = 3                                       # p = # of ADFs
    '''
    '''
    input_list = [(1,2), (2,3), (1,4), (3,4)]
    n = 4
    '''
    '''
    input_list = [(2,1), (2,4), (5,3)]
    n = 5
    '''
    theater= [0]*(n**2)
    # theater is p x p list
```

Secondly, let's convert a list of pairs into an adjacency list as shown in Figure 2. We will implement an adjacency list as a dictionary as we have seen in Q1. In line 25, we will start with an empty dictionary. Next line 28 to 30, we will fill the keys in the dictionary based on n given to us as an input. From line 32 to 35, we will iterate through each pair and look at both elements of the pair and fill respective friends' lists. From line 37 to 44, we figure out who would be the starting node or who would sit on the first on based on the number of direct friends. First in line 37, we give the first element in the adjacency the honor to sit on the first row. This may seem arbitrary, but the first element will sit on the first row, if everyone else has the exact same number of friends. In line 42, if the current node has more friends than the current max_key, we reassign max_key to the current node. After the for loop terminates, we know who will sit on the first row.

Thirdly, let's discuss the initialization step of the algorithm as shown in Figure 3. In line 48, we will make a list that will tell if an element has been seated or not. For example, if $ADF7$ has been seated, $seated\_check\_list[6] == 1$. This will be useful as our while loop will continue until every element in this array is 1. In line 50, we will use queue as our data structure as we have in Q1. In 53, we will start with the starting node or max_key by putting it in the queue, and we note the max_key bas been seated in the first row or about to be seated in the first row. In order to make sure where to direct a node for an appropriate seat, we keep track of current row called cur_row and current column called cur_column. Note that we also have maximum column, which will be used when we direct a node

Figure 2: ADF Seating - Create Adjacency List

```
24          ### input_list will be converted into a adjacency list
25          adjacency_list = {}
26          #empty dictionary
27
28          for i in range(1, n+1):
29              #dictionary with p keys
30              adjacency_list[i] = []
31
32          for p in input_list:
33              #complete adjacency list
34              adjacency_list[p[0]].append(p[1])
35              adjacency_list[p[1]].append(p[0])
36
37          max_key= list(adjacency_list.keys())[0]
38          for key, value in adjacency_list.items():
39              #sort the adjacency list
40              value.sort()
41
42              if len(adjacency_list[max_key]) <len(value):
43                  max_key=key
44                  #max key will the starting node at the first row
45
46          ### complete constructing adjacency list
```

disconnected from the first starting node to be seated in a certain column in the first row. In line 58, the max_key sits on $(0,0)$.

Figure 3: ADF Seating - Initialization

```
48          seated_check_list = [0]*n
49          #check if an element has been seated
50          q = deque()
51
52          ##initialize seating
53          q.append(max_key)
54          seated_check_list[max_key-1] = 1
55          #max_key takes a seat
56          cur_row = 0
57          cur_column= max_column = 0
58          theater[n*cur_row + cur_column] = max_key
```

Now let's discuss the core of the algorithm. In line 61, while loop indicates that the following algorithm will continue to execute until everyone is seated. Note that the inside of the while loop is sectioned into two parts: 1) try: from line 64 to 77 and 2) $except\ IndexError$: from line 78 to 87. The first section from line 64 to 77 will execute as long as there is something in the queue or said differently as long as we have a node connected to the current starting node. The second section will run if 1) there is at least one node that has not seated and 2) the node is disconnected from any node that has already taken a seat.

Let look into the first section from line 64 to 77. First we'll pop the first item in the queue and assign it as the current

node. In line 65, we increment the current row because the for loop below completes a breadth search. Said differently, since the starting node has taken a seat, we need to look for ADFs who will seat on the second row. In line 67, we will iterate through every friend of the current node. Line 69 checks whether if a node has already taken seat, if so we don't need to take any action (line 76 to 77). If he or she has not taken a seat, the algorithm will assign a seat based on the current row and current column in line 70. We will mark that this ADF has taken a seat in 71. We increment the current column because the next person will sit on the same row and right of the current node. Lastly, we will add the current node to the queue in line 75 so that when we are done with the current level, we will visit the friend list of the current node.

Figure 4: ADF Seating - Seating

```
61          while 0 in seated_check_list:
62          #while loops until everyone is seated
63              try:
64                  current_node = q.popleft()
65                  cur_row+= 1
66                  #increment after seating one row (one BFS level)
67                  for j in adjacency_list[current_node]:
68
69                      if seated_check_list[j-1] ==0:
70                          theater[n*cur_row + cur_column] = j
71                          seated_check_list[j-1] = 1
72                          cur_column +=1
73                          max_column += 1
74                          #next seats in the same row, next column
75                          q.append(j)
76                      else:
77                          pass
78              except IndexError:
79              #accounting for disconnected node / graph when queue is empty
80                  for key, value in adjacency_list.items():
81                      if seated_check_list[key-1]==0:
82                          q.append(key)
83                          seated_check_list[key-1] = 1
84                          cur_row = 0
85                          cur_column = max_column
86                          theater[n*cur_row + cur_column] = key
87                          break
```

Let's talk about the second section from line 78 to 87. When there is at least a node that has not taken a seat (0 exists in $seated\_check\_list$) and the queue is empty, popping the queue will return an IndexError. If this happens, we will go through the keys in the adjacency list, and pick the first ADF that has not taken a seat and push it to the queue and mark it as seated. This person can seat anywhere, but for our algorithm we will assign it to the first row. Also we will change the current column to maximum column so that all nodes that are connected to the "new" starting node will seat "right" of the previous group. With a node in the queue, we can go back to the beginning fo the while loop.

The while loop will terminate when everyone has taken a seat.

4 Part (b): Proof of Correctness Idea

In order to prove that our algorithm, we need to show the seating the algorithm produces is indeed admissible. A seating needs to satisfy all three conditions in order to be admissible. First, we know that algorithm will always have at least one person sit in the first row as long as an input size ≥ 1. Therefore, we know that one condition always is met.

In order to prove the algorithm's seating satisfies two other conditions, we can frame this question as constructing a BFS tree T for the graph G. The graph G is the list of pairs of ADFs given to us as an input to the algorithm. In order to prove that the algorithm satisfies the distance compatibility from the starting node, we will use Lemma 4.1 presented in page 80 of the textbook. In order to prove that the algorithm satisfies the condition that requires friends are seated in the same row or in adjacent rows, we will use Lemma 4.2 presented in page 81 of the texbook.

Lemma 4.1. *For each $i \geq 1$, layer L_j produced by BFS consists of all nodes at distance exactly j from s. There is a path from s to t if and only if t appears in some layer.*

Lemma 4.2. *Let T be a breadth-first search tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge of G. Then i and j differ by at most 1.*

5 Part (b): Proof Details

The proof of correctness of the algorithm will be by contradiction. We will assume an output that violates each condition, then show both situations lead to contradiction.

- Suppose there are starting node s and a connected node t. This means that a path exists between s and t. Suppose s is j distance away from s and is at the layer $\neq j$.Based on the BFS algorithm, we know that L_1 consists of all nodes that are neighbors of s. Also assuming that we have defined layers $L_1, ...L_{j-1}$, we know L_j consists of all nodes that do not belong to an earlier layer and that have an edge to a node in L_{j-1}. It follows that node s is either layer earlier than L_j or later than L_j. It cannot be at L_j. This is a contradiction.
- Our algorithm does not implement specific lines of codes to make sure friends are seated in the same row or an adjacent row. In other words, our BFS should take care of this condition automatically. Lemma 4.2 is directly related to this. The lemma says if an edge exists betwen (x, y), the distance between x and y are at most 1. Please refer to the proof details in the textbook.

6 Part (b): Runtime Analysis

In order to analyze the runtime of our algorithm, we will go through Figure 2, 3 and 4 step by step. In Figure 2, for loop in line 28 takes $O(n)$ where n is the number of ADFs. For loop in line 32 takes $O(m)$ whehre m is the number of pairs or the number of edges. The for loop in line 38 takes $O(n)$. We can assume the operations such as comparison and accessing a list take constant time of $O(1)$. The overall time for Figure 2 is $O(n + m)$ or linear to the input.

The runtime for Figure 3 is straightforward. All the operations stated here is $O(1)$.

Now let's look at Figure 4. Our algorithm is structurally similar to the algorithm presented in page 90 of the textbook. According to the textbook, we can say that the algorithm is $O(n + m)$.